The tool of thought for expert programming

Release Notes

Version 12.1

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2009 by Dyalog Limited.

All rights reserved.

Version 12.1.0 produced on 2009/11/06

First Edition October 2009

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.
UNIX is a trademark of X/Open Ltd.
Windows, Windows Vista, Visual Basic and Excel are trademarks of Microsoft Corporation.

All other trademarks and copyrights are acknowledged.

Contents

CHAPTER 1 Introduction **1**

Summary 1
New Functions 1
Faster Primitive Functions 2
Parallel Execution (multi-cpu) 4
Component File Journaling and CheckSum 4
User Commands 5
Standalone Executables 6
.Net Controls 8
NetControl ..... Object 10
APLVersion ..... Property 13
RadiusMode ..... Property 14
Changes to Keyboard Support 15
Enhanced Editor 20
APLFormatBias Parameter 31
Miscellaneous 32
System Requirements 33
Interoperability and Compatibility 34

CHAPTER 2 XML Import and Export **37**

Introduction 37
XML Processing 38
Conversion from XML 42
Conversion to XML 44

CHAPTER 3 Language Enhancements **45**

New and Revised Primitive & System Functions 45
- Table: ..... R←⍪Y 46
- I-Beam: ..... R←{X}(A⌶)Y 47
- Syntax Colouring: ..... R←200⌶Y 48
- Number of Threads: ..... R←1111⌶Y 49
- Parallel Execution Threshold: ..... R←1112⌶Y 49
- Thread Synchronisation Mechanism: ..... R←1113⌶Y 49
- Memory Manager Statistics: ..... R←2000⌶Y 50
- Export to Memory: ..... R←2100⌶Y 50
- Expunge Object: ..... {R}←⎕EX Y 51
- File Check and Repair: ..... R←{X} ⎕FCHK Y 53
- File Properties: ..... R←X ⎕FPROPS Y 54

Size of Object: .... R←⎕SIZE Y 58
XML Convert: .... R←{X} ⎕XML Y 59
Outer Product: .... {R}←X∘.gY 67

Index 69

CHAPTER 1

Introduction

Summary

Dyalog APL Version 12.1 provides the following new features and enhancements:

- New *Table* function (monadic ⍪)
- New *I-Beam* function
- New system function ⎕XML.
- Faster primitive functions
- Parallel execution on multi-cpu systems
- User Commands
- The facility to export a workspace as a standalone `.exe`
- Enhanced Component File Journaling
- Enhanced Editor
- New NetControl object to embed .Net controls in the Dyalog GUI.
- Changes to Keyboard Support
- RadiusMode property to obtain perfectly round circles

New Functions

Table function

The new *Table* function (monadic ⍪) turns any array into a 2-dimensional matrix.

I-Beam

The *I-Beam* function (⌶) provides access to certain specialised system services.

XML

The ⎕XML system function converts an XML character vector into an APL array and vice-versa. As a built-in system function it is naturally much faster than an equivalent defined function written in APL.

File Check and Repair

The ⎕FCHK system function validates and repairs a component file.

Faster Primitive Functions

A number of primitive functions have been modified to deliver significantly improved performance in Version 12.1.

Boolean Scan and Reduce

Many scans and reductions on Boolean arrays perform a lot faster in Version 12.1. These include the following derived functions:

```
∧/ ∨/ =/ ≠/ +/ ∧\ ∨\ =\ ≠\ <\
```

Many inner products where one of the above functions is used as the left function operand are also significantly faster, as they use Boolean reduction.

Grade up and down

The performance of Grade Up and Grade Down has been improved, especially for *small-range* vectors. A ***small-range*** array is a simple integer or character array where the difference between its minimum and maximum value is less than twice its number of items.

In addition, the following expressions have been implemented as idioms for small-range vectors and narrow small-range matrices (less than 8 bytes per row):

```
{⍵[⍋⍵]} {⍵[⍒⍵]} {⍵[⍋⍵;]} {⍵[⍒⍵;]}
```

Note that it is only these precise expressions that have been optimized by idiom recognition; expressions such as X[⍋X;] are not recognized as idioms.

Compress

Compress on the leading axis has been tuned

Matrix Multiplication

Matrix multiplication (+.×) has been rewritten and is substantially faster in Version 12.1.

Basic Arithmetic Functions

The basic arithmetic dyadic functions (+ - ×) have been rewritten in assembler (x86 platforms) using SSE2 operations where possible.

The new code is implemented in Dyalog APL Version 12.1 for Windows (32 and 64-bit) and for Linux (32 and 64-bit).

On all other platforms and on x86 processors that do not support SSE2, e.g. Intel processors predating Pentium 4 (2001) and AMD processors predating K8 (2003), the previous (Version 12.0) code applies.

Indexing

Simple indexing and indexed assignment have been completely rewritten. They consume much less memory and many special-cases are optimized.

Set Functions

The set functions (index of, membership, union, intersect, without, unique) now use improved algorithms for certain data types:

- Small-range arrays, including all single-byte data types, use a table look-up algorithm that is considerably faster than the old hashing algorithm.
- In cases where one argument has a small number of elements, the functions now use an optimised linear search mechanism.
- Outer products (e.g. ∘.∩) use retained hash or index tables where relevant.

Transpose

Transpose has been optimized.

Parallel Execution (multi-cpu)

If your computer has more than one cpu or is a multi-core processor, many scalar dyadic functions (÷ ≥ = ≤ ⍟ | ! ○ ∨ ∧) now execute in parallel in separate system threads each running on a separate cpu or core, when the argument size exceeds a configurable limit, the parallel execution threshold.

For example, by default if you have a 4-cpu computer and execute an expression such as (A÷B) where A and/or B contain more than 32768 elements, Dyalog will start 4 separate threads running on the 4 separate cpus each performing the division on ¼ of the elements of the array(s) and simultaneously creating the corresponding ¼ of the result array. The threads are only started once, and reused for subsequent multi-threaded operations.

The maximum number of cpus to use can be controlled using 1111 ⌶, and the parallel execution threshold) is changed using 1112 ⌶. These “tuning” I-beams should be considered experimental, and may be changed or replaced by automated tuning in a future release.

Note that the functions (+ - ×) are NOT executed using multiple cpus. This is because a single cpu is able to perform these operations so fast that memory is barely able to keep up. Tests show that multi-threading on separate cpus does not increase the speed of these operations.

Component File Journaling and CheckSum

Version 12.1 provides enhanced Journaling for Dyalog APL component files and includes the facility to *check sum* components.

The latter is highly recommended as it enables component files to be repaired using the new system function ⎕FCHK. However, c**omponent files written with Checksum enabled cannot be read by versions of Dyalog APL prior to Version 12.1.**

In addition to the level of Journaling provided in Version 12.0 (now called Level 1 Journaling), two further levels are introduced.

Level 2 journaling provides protection not just against the possibility that the APL process terminates abnormally, but that the Operating System itself fails. However, a damaged component file must be explicitly repaired using the new system function ⎕FCHK which will repair any damaged components by rolling them back to their previous states.

Level 3 provides the same level of protection as Level 2, but following the abnormal termination of either APL or the Operating System, the rollback of an incomplete update will be automatic and no explicit repair will be needed.

Higher levels of Journaling inevitably reduce the performance of component file updates.

For further information, see ⎕FPROPS and ⎕FCHK.

User Commands

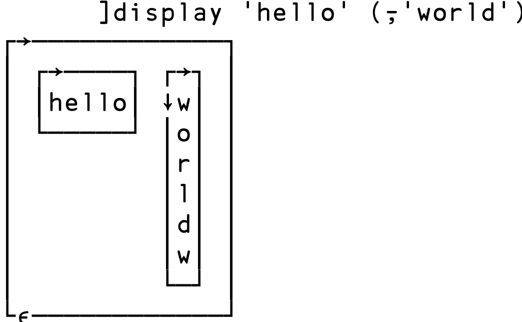
Version 12.1 provides an easy mechanism to define User Commands.

User commands are developer tools, written in APL, which can be executed without having to explicitly copy code into your workspace and/or save it in every workspace in which you want to use it.

A User Command is a name prefixed by a closing square bracket, which may be niladic or take an argument. A User Command executes APL code that is typically stored somewhere outside the current active workspace.

By default, the existing SPICE command processor is hooked up to the user command mechanism, and a number of new SPICE commands have been added. For example:

```
      ]display 'hello' (⍪'world')
```

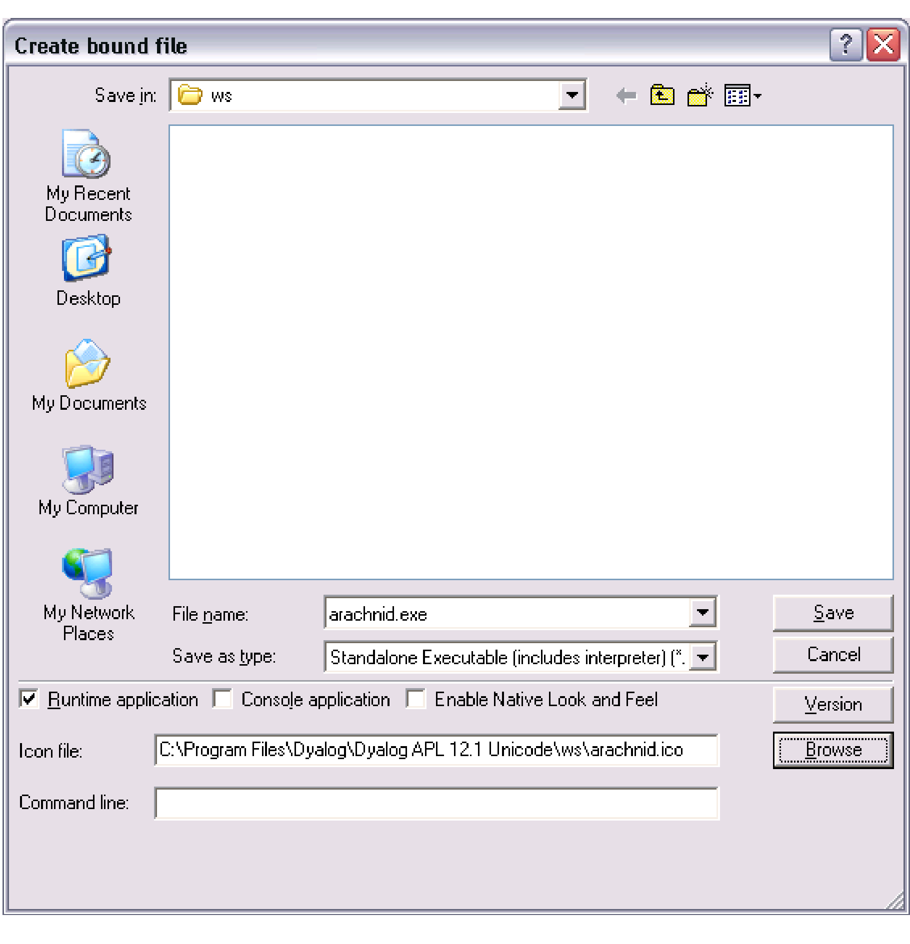


The implementation of User Commands is very simple: If a line of input begins with a closing square bracket (]), and there exists a function by the name ⎕SE.UCMD, then the interpreter will call that function, passing the input line (without the bracket) as the right argument.

To add a user command, drop a new Spice command file in the folder SALT\Spice.

Standalone Executables

In Version 12.1 it is possible to export a workspace as a single `.exe` file that includes a copy of the Dyalog APL interpreter. This allows you to implement and deploy a stand-alone Dyalog APL application as a single `.exe` file.



Note however, that if your application requires.Net support you must to deploy 2 Dyalog dynamic link libraries, namely `dyalognet.dll` and the appropriate *Bridge dll* according to the following table.

	32-bit	**64-bit**
Unicode Edition	`bridge121_unicode.dll`	`bridge121-64_unicode`
Classic Edition	`bridge121.dll`	`bridge121-64.dll`

This facility has been added to the *Save as type* drop-down menu on the Create bound file dialog box, accessed via File->Export from the Session menu bar. The operation to export the current active workspace as a stand-alone executable may also be actioned programmatically using `2110 ⌶`.

.Net Controls

Whereas previously it was possible to use Microsoft .Net GUI by creating a .Net Form and populating it with .Net controls, it was not possible to mix .Net GUI components with the Dyalog APL GUI.

In Version 12.1, the new `NetControl` object allows you to include any control that derives from System.Windows.Forms.Control as a child of a Dyalog APL GUI container such as a Form, SubForm, or Group.

```
⎕using←'System.Windows.Forms,system.windows.forms.dll'
'f'⎕wc'Form' ('Coord' 'Pixel')
'f.wb'⎕WC'NetControl' 'WebBrowser' ('Size' f.Size)
f.wb.GoHome ⍝ Display user's home page
```

Many .Net controls can also be used as input objects for the Dyalog Grid.

In addition classes defined within the workspace that derive from System.Windows.Forms.Control can also be embedded into the Dyalog GUI.

NetControl Object

Purpose This object allows you to embed .Net Controls in the Dyalog GUI.

Parents Form, Grid, Group, PropertyPage, SubForm

Children NetClient, OLEClient, Timer

Properties Type, Posn, Size, Coord, ClassName, Attach, MethodList, ChildList, EventList, PropList

Events (None)

Methods (None)

In principle, you may use the NetControl to embed any class that derives from System.Windows.Forms.Control (from system.windows.forms.dll), including derived classes written in Dyalog APL.

To load a particular .Net control, the appropriate .Net Assembly must be specified in ⎕USING; otherwise the expression will cause a LIMIT ERROR. For example, to load one of the standard .Net controls:

```
⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
```

The ClassName property specifies the name of the .Net control to be instantiated and to which the new object named by the left argument of ⎕WC is to be connected. ClassName may only be specified by ⎕WC.

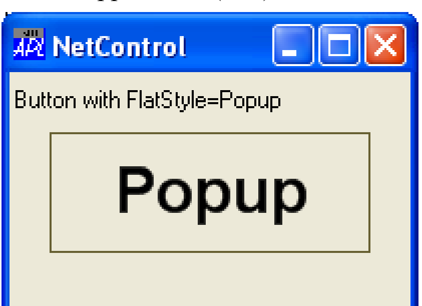
Once you have created an instance of a particular NetControl, the properties, events and methods it supports may be obtained using ⎕NL. These are the properties, events and methods defined for the control by its author. The “Dyalog” properties listed above, are not reported by ⎕NL, but take precedence over (i.e. mask) any members of the same name that may be exposed by the class itself.

The following example illustrates the use of the Button class. In this case, the FlatStyle property of the button is set to “Popup”. This gives the button a flat appearance until the mouse is hovered over it, when its appearance it changes to 3-dimensional.

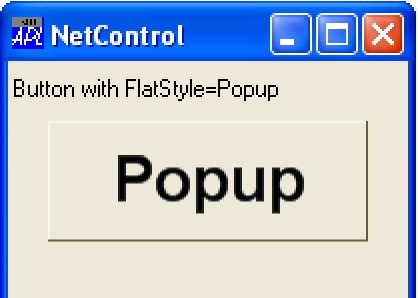
```
⎕USING←'System'
⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
⎕USING,←⊂'System.Drawing,system.drawing.dll'
```

```
 an←⎕NEW FontFamily(⊂'Arial')
 myfont←⎕NEW Font(an 24 FontStyle.Bold GraphicsUnit.Point)

 'f'⎕WC'Form'('Coord' 'Pixel')('Size' 120 200)
 f.Caption←'NetControl'
 'f.l'⎕WC'Label' 'Button with FlatStyle=Popup'(2 2)


'f.b'⎕WC'NetControl' 'Button'('Size' 60 160)

 f.b.⎕nl -2
AutoSizeMode  DialogResult  AutoEllipsis  AutoSize
BackColor  FlatStyle  FlatAppearance ...

 f.b.⎕nl -3
BeginInvoke  BringToFront  Contains  CreateControl
CreateGraphics  CreateObjRef  Dispose  DoDragDrop ...

 f.b.⎕nl -8
DoubleClick  MouseDoubleClick  AutoSizeChanged
ImeModeChanged  BackColorChanged  ...

 f.b.Text←'Popup'
 f.b.Font←myfont

 f.b.(FlatStyle←FlatStyle.Popup)
```

Normal appearance (Flat)



Appearance when mouse over



In most cases, you may use a NetControl in the cells of a Grid object. Unless you specify otherwise, using the InputProperties property of the Grid, the *default* property of the NetControl will be associated with the corresponding element of Values. The following example illustrates the use of a TextBox control. In this example, the CharacterCasing property of the TextBox is set to Upper, causing all text to be converted to upper-case.

```
⎕USING←'System'
⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
⎕USING,←⊂'System.Drawing,system.drawing.dll'

an←⎕NEW FontFamily(⊂'Arial Narrow')
myfont←⎕NEW Font(an 11 FontStyle.Bold GraphicsUnit.Point)

'f'⎕WC'Form' ('Coord' 'Pixel')('Size' 130 500)
f.Caption←'Grid using .Net TextBox Control'

'f.g'⎕WC'Grid'('Posn' 0 0)f.Size
f.(ShowInput TitleWidth)) ← 1 0

'f.g.tb'⎕WC'NetControl' 'TextBox'
f.g.tb.Font←myfont
f.g.tb.(CharacterCasing←CharacterCasing.Upper)

f.g.Input←'f.g.tb'

wds←'All' 'TeXt' 'Is' 'Changed' 'to' 'Upper' 'casE'
wds,← 'ακομα' 'kai' 'τα' 'Ελληνικα'

f.g.Values←5 5⍴wds
```

Grid using .Net TextBox Control

A	B	C	D	E
ALL	TEXT	IS	CHANGED	TO
UPPER	CASE	AKOMA	KAI	TA
ΕΛΛΗΝΙΚΑ	ALL	TEXT	IS	CHANGED
TO	UPPER	CASE	AKOMA	KAI
TA	ΕΛΛΗΝΙΚΑ	ALL	TEXT	IS

Implementation note: The instance of the .Net control is actually placed inside an instance of the .Net class `System.Windows.Forms.ContainerControl`.This `ContainerControl` is then embedded in the Dyalog parent, such as a Form. This "extra level" should have no affect on how the control is used or on how it behaves.

APLVersion Property

Applies to Root

Description

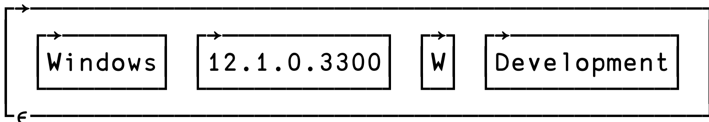
This is a read-only property that provides information about the Version of Dyalog APL that you are using. It is a 4-element vector of character vectors as described in the table below.

Note: In future releases these values may change, be removed, or new ones added.

Index	Description	Possible Values
[1]	Target Environment	Windows Windows-64 Windows Mobile Linux Linux-64 AIX AIX-64 Solaris Solaris-64
[2]	Version Number	
[3]	Version Type	W : Windows S : Server (terminal) version Wine : GUI version running under WINE M : Motif P : PocketAPL
[4]	Program Type	Development Runtime DLL

Example:

```
      ]display '.' ⎕WG'APLVersion'
```



RadiusMode Property

Applies to Circle, Root

A perfectly round circle can only be drawn if the diameter is an odd number of pixels. The RadiusMode property specifies whether or not a circle is adjusted by a single pixel, if necessary, so as to appear perfectly round.

If RadiusMode is 1 or `¯1`, and the diameter is an even number of pixels, the circle is actually drawn with a diameter of 1 pixel more or less than specified. If RadiusMode is 0 (the default), no such adjustment is made.

RadiusMode may be set on the Root object to be inherited by all Circle objects.

Changes to Keyboard Support

Version 12.1 Unicode Edition ships with three sets of keyboard layouts. By default, one keyboard from each of these sets (corresponding to your language or locale) will be installed:

Ctrl: Keyboards named *locale – Dyalog Ctrl* (e.g. *UK – Dyalog Ctrl*). These keyboards use what has become the "classic" Dyalog keyboard, using the Ctrl key to enter APL symbols.

The "Ctrl" keyboards shipped with Version 12.1 are slightly different from earlier versions (with the exception of the Italian keyboard, which was ahead of its time) in that the APL symbols ⊂⊃∩∪ can also be generated using Ctrl+Shift in combination with Z,X,C,V. This is useful in applications which use Ctrl+Z,X,C,V for Undo, Cut, Copy and Paste. By default, the APL session will now use Ctrl X,C,V for Cut, Copy and Paste.

AltGr: Keyboards named *locale – Dyalog AltGr* (e.g. *UK – Dyalog AltGr*), which are very similar to the above, except that the AltGr key (available on some keyboards) is used to select APL symbols. Note that Alt+Ctrl can be used as an alternative to the AltGr key.

IME: An *Input Mode Editor* (which is a special kind of keyboard handler) named *Dyalog IME* supports configurable keyboards which are defined using the same Dyalog Input (.DIN) files as the Classic Editions. By default, the IME will be configured with a Ctrl layout. The new IME does not have the *APL* On/Off button that previous versions had – the intention is that it will be activated using standard Windows mechanisms.

I-Beam (⌶) has been added to all keyboards as Ctrl+Shift+I or AltGr+Shift+I.

Which Keyboard Should I Use?

The three families of keyboards have the following advantages and disadvantages, depending on which application you use them with:

Ctrl keyboards are familiar and likely to be the first choice for anyone with experience in using Dyalog APL. The keyboards work well in the APL session, but if you use them with other common Windows applications, many of the APL symbols cannot be entered because applications use key combinations like Ctrl+A to select all text or Ctrl+S to save the current document.

AltGr keyboards move the APL symbols onto keys which are less frequently used as shortcuts. However, there is generally only one AltGr key on a keyboard, so typing is a little less convenient. Also, some applications are starting to use AltGr-based shortcuts. For example, Google Desktop uses AltGr+G as a global hotkey.

The **IME**, which had been introduced with the version known as Dyalog.Net (between versions 10.0 and 10.1) as a tool for editing APL Script files, was retired in version 12.0 because it was believed that the **Ctrl** and **AltGr** keyboards would now be sufficient. However, the IME is different from "normal" keyboards in that any keystroke which is defined as producing a symbol becomes invisible to the underlying application, which only sees the character which is generated. Thus, if you use the Dyalog IME with Microsoft Word, Ctrl+A will not select all the text in your document, but always produce the symbol ⍺. Of course, this means that you can no longer use hotkeys based on Ctrl in Word when the IME is active.

Recommended Strategies

Dyalog recommends one of the following two strategies:

AltGr: If you are comfortable with the **AltGr** keyboard and you don't have many situations where you need to type APL characters into an application in which AltGr is used for hotkeys, set your *locale - Dyalog AltGr* keyboard up as the default keyboard on your machine, and it should work well in all applications. You can activate the IME using the Windows Language Bar if you occasionally have a problem with application hotkeys making it hard to enter APL symbols.

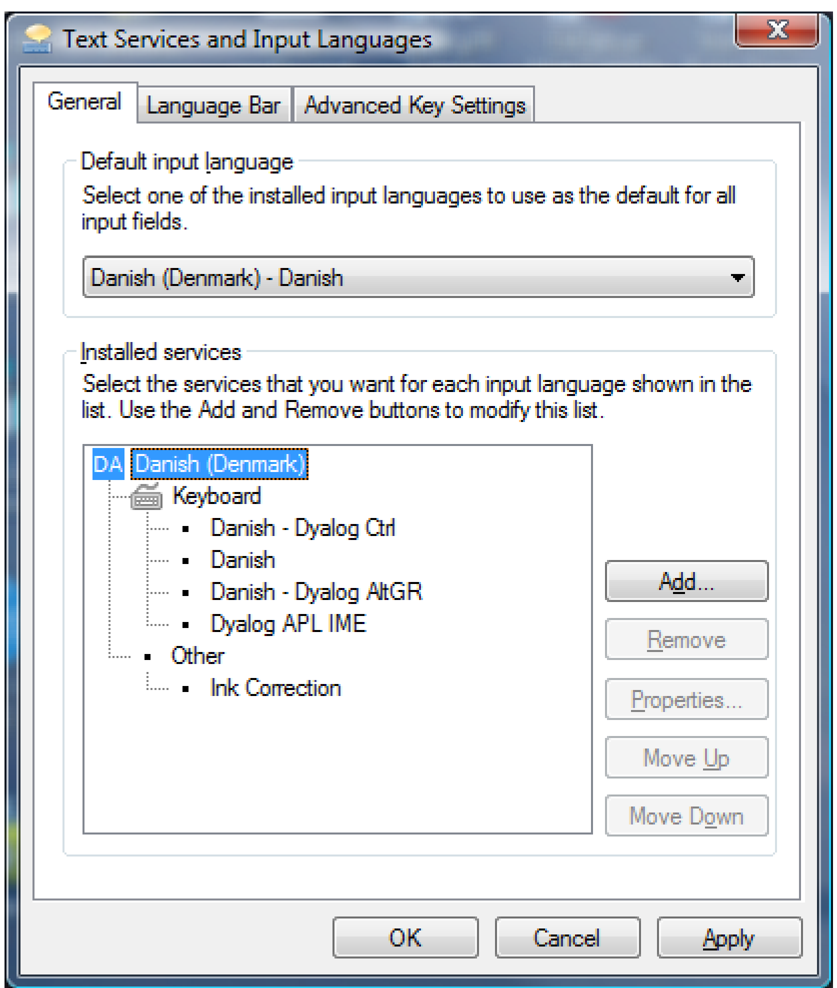
Ctrl+IME: If you find that the **Ctrl** layout is more attractive, you can use the **Ctrl** keyboard most of the time, and switch to the IME if you need to enter APL symbols into an application which uses Ctrl for hotkeys. If you need to switch frequently, we recommend that you use Control Panel to set up a hotkey to toggle between keyboards (and possibly remove the AltGr keyboard to reduce the number of keyboards that you will be toggling between).

Configuring keyboards

Keyboards can be configured using:

> *Control Panel/Regional and Language Settings/Keyboards and Languages*, then press the *Change Keyboards* button.

This displays the following dialog box:



You can also bring this dialog box up by right clicking on the language bar and selecting Settings. You can use this dialog box to remove unwanted keyboards (and add them back again). Use the *Advanced Key Settings* tab to define shortcuts to switch between keyboards.

If you are going to be switching between keyboards frequently (for example, when writing documentation), you can "restore" the *Language Bar* so that it is easier to see which keyboard is selected (when minimized, the *Language Bar* only shows a small picture of a keyboard on the taskbar). When restored, it looks like this:

Configuration Options

A few configuration options are worth mentioning.

In APL, on the *Keyboard Shortcuts* tab of the *Options/Configure* dialog, a new checkbox labeled *Use Ctrl X,C,V for Clipboard* allows you to decide whether APL will always use these shortcuts, regardless of what other shortcuts are defined.

The IME has a couple of registry entries which can be used to configure it – in the registry section HKEY_CURRENT_USER\Software\Dyalog\IME:

`Translate Table` Names the .din file to be used to define the input translate table. This is the same as the `aplk` parameter used by Classic APL systems. By default, the DIN file corresponding to your locale should have been automatically selected.

`WantsSpecialKeys` Lists applications for which the IME will generate the characters defined in the "special functions" section of the DIN file. The IME will always do this when the application is dyalog.exe or dyalogrt.exe. By default, this key is initialized to "putty", so that special keys are processed when using the PuTTY terminal emulator. The default translate tables provided with Unix when this is used to run a terminal session which is connected to and APL system which is using a translate table which recognizes the symbols.

Troubleshooting

Problem: **No Dyalog keyboards installed, or keyboards were installed for the wrong language**

Solution: Report the problem to support@dyalog.com, providing the value of the registry key "1" in *HKEY_CURRENT_USER\Keyboard Layout\Preload* (this should be an 8-digit number, for example "00000406" for Denmark).

Problem: **Keyboard layout changes unexpectedly while typing**

Solution: Check whether a keyboard shortcut is defined to switch between keyboard layouts: Right click on the Windows *Language Bar*, select *Settings*, and select the *Advanced Key Settings* tab.

Enhanced Editor

The Editor has been enhanced to improve the ease with which individual functions and classes can be edited.

Functions

For single functions, the most obvious improvement is the *outlining* feature. Outlining identifies the blocks of code within control structures, and allows you to collapse and expand these blocks so that you can focus your attention on particular parts of the code

The picture below shows the result of opening the function ⎕SE.cbtop.TB_POPUP.

```
      )ed ⎕SE.cbtop.TB_POPUP
```

```
Editor - [⎕SE.cbtop.TB_POPUP]
File  Edit  Refactor  Window  View
[0]   {FLAG}TB_POPUP MSG;TC
[1]   ⍝ Popup menu on Session ToolControl
[2]   :If 0=⎕NC'FLAG'
[3]       :If (2=5⊃MSG) ⍝ Right mouse button ?
[4]           'popup.captions'⎕WS'Checked'ShowCaptions
[5]           'popup.list'⎕WS'Checked'('FlatList'≡Style)
[6]           'popup.multiline'⎕WS'Checked'MultiLine
[7]           ⎕DQ'popup'
[8]       :End
[9]   :Else
[10]      TC←'bandtb1.tb' 'bandtb2.tb' 'bandtb3.tb' 'bandtb4.tb'
[11]      :Select FLAG
[12]      :Case 1
[13]          ShowCaptions←~ShowCaptions
[14]          TC ⎕WS¨⊂'ShowCaptions'ShowCaptions
[15]      :Case 2
[16]          Style←(1+'FlatList'≡Style)⊃'FlatList' 'FlatButtons'
[17]          TC ⎕WS¨⊂'Style'Style
[18]      :Case 3
[19]          MultiLine←~MultiLine
[20]          TC ⎕WS¨⊂'MultiLine'MultiLine
[21]
[22]      :End
[23]  :End
Function | Last saved by: Dyadic: 29 October 1998 19:15 | Pos: 0/24,1
```

Notice that the various control structure blocks are delineated by a treeview diagram.

- When you hover the mouse pointer over one of the boxes that mark the start of a block , the line marking the extent of that block becomes highlighted, as shown above.
- If you click on a ⊟ box, the corresponding section collapses, so that only the first line of the block is displayed, as shown below.
- If you click on a ⊞ box, the corresponding section is expanded.

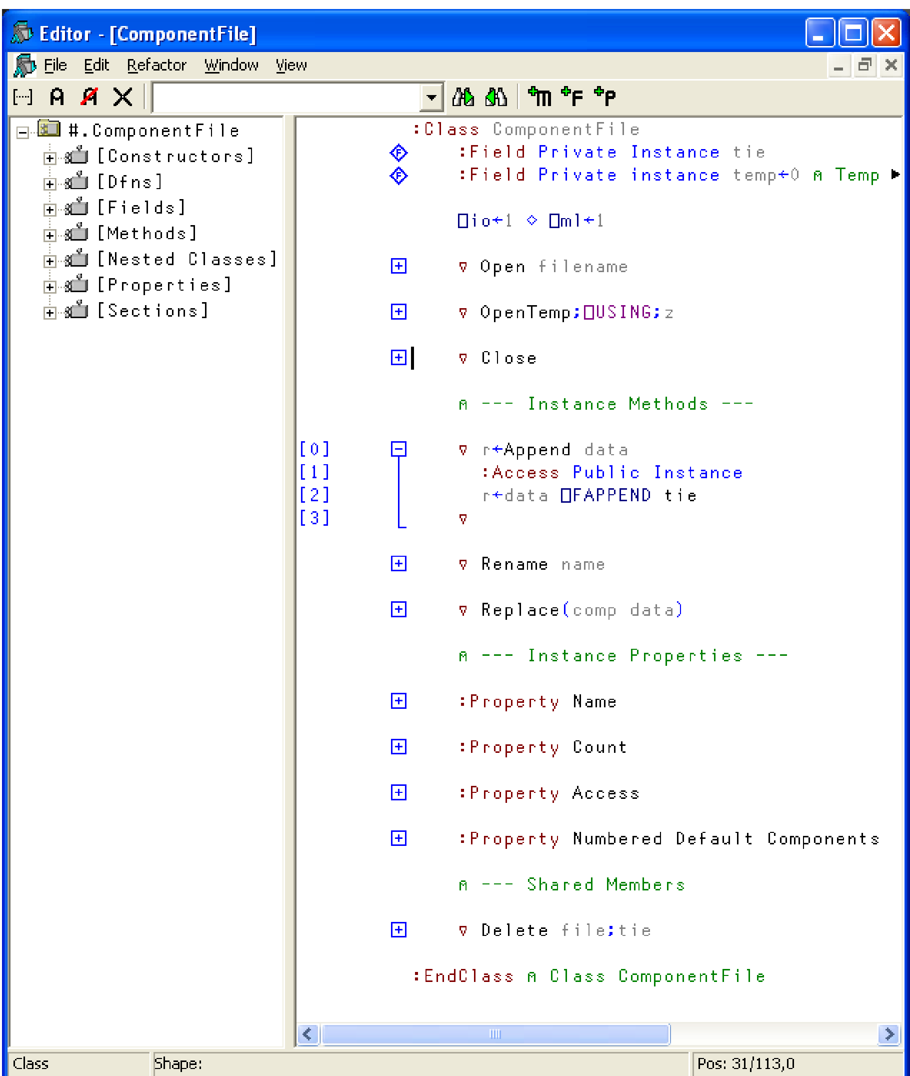
```
Editor - [⍟SE.cbtop.TB_POPUP]
File  Edit  Refactor  Window  View

[0]        {FLAG}TB_POPUP MSG;TC
[1]        ⍝ Popup menu on Session ToolControl
[2]        :If 0=⎕NC'FLAG'
[3]            :If (2=5⊃MSG) ⍝ Right mouse button ?
[9]        :Else
[10]           TC←'bandtb1.tb' 'bandtb2.tb' 'bandtb3.tb' 'bandtb4.tb'
[11]           :Select FLAG
[12]           :Case 1
[13]               ShowCaptions←~ShowCaptions
[14]               TC ⎕WS¨⊂'ShowCaptions'ShowCaptions
[15]           :Case 2
[16]               Style←(1+'FlatList'≡Style)⊃'FlatList' 'FlatButtons'
[17]               TC ⎕WS¨⊂'Style'Style
[18]           :Case 3
[19]               MultiLine←~MultiLine
[20]               TC ⎕WS¨⊂'MultiLine'MultiLine
[21]
[22]           :End
[23]       :End

Function | Last saved by: Dyadic: 29 October 1998 19:15 | Pos: 0/24,1
```

Editing Classes

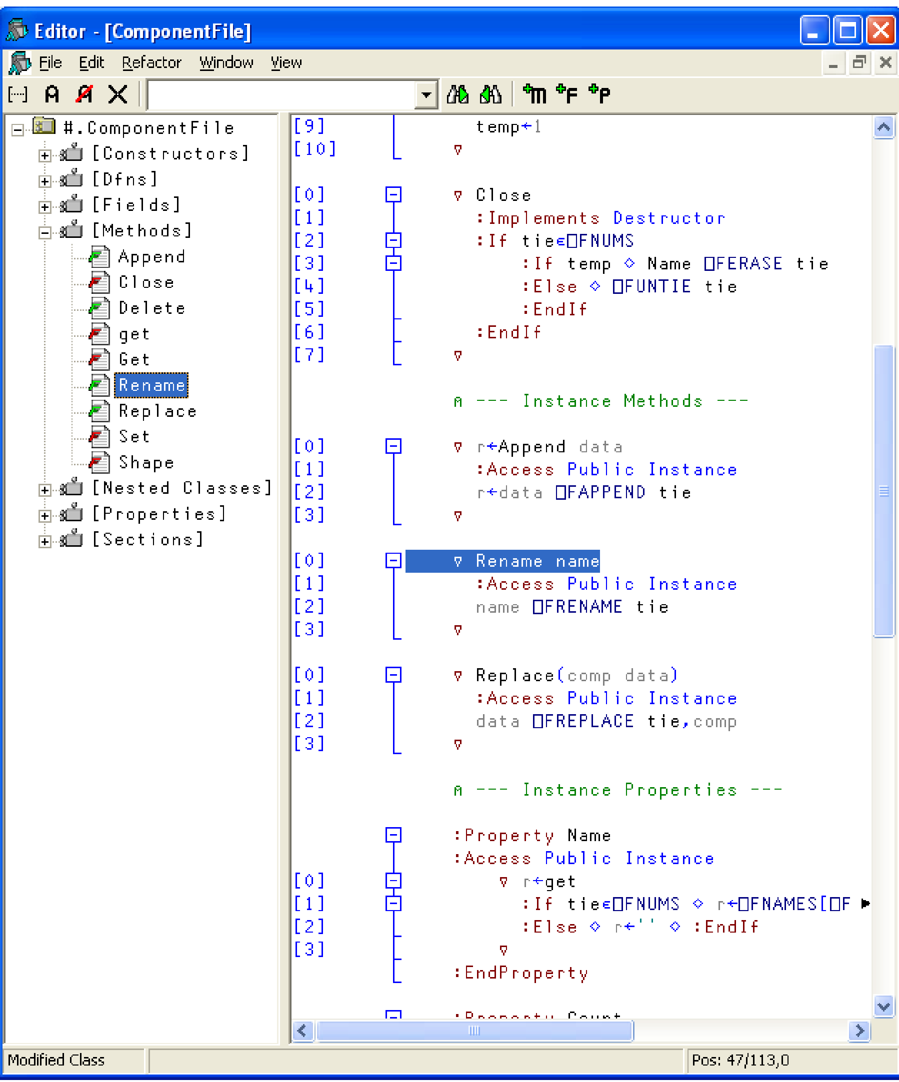
The picture below shows the result of opening the ComponentFile class. Notice how each function is delineated separately and that each function is individually line-numbered.

```
      )ed ComponentFile
```

The outlining feature really comes into its own when editing classes because you can collapse and expand whole functions. The picture below shows the effect of collapsing all but the `Append` method.

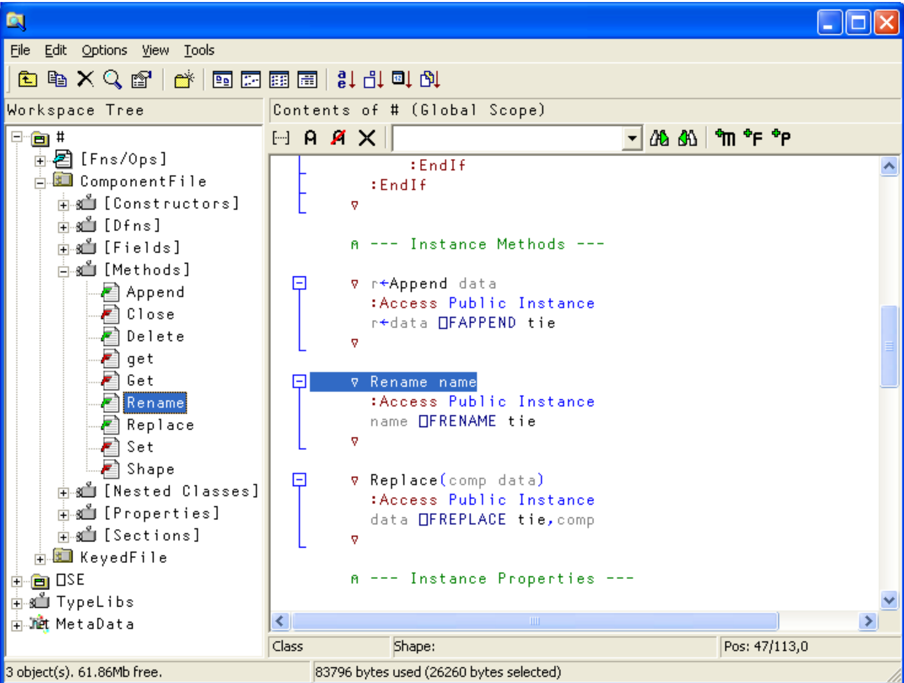


When you edit a class, a separate treeview is optionally displayed in the left pane to make it easy to navigate within the class. When you click on a name in the treeview, the editor automatically scrolls the appropriate section into view (if necessary) and positions the edit cursor at its start. The picture below illustrates the result of opening the [Methods] section and then clicking on `Rename`.

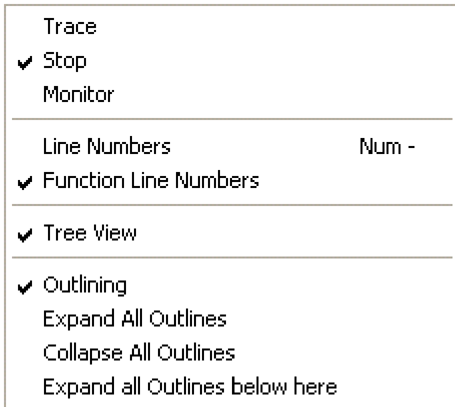


Editing from the Workspace Explorer

If you double-click on the name of a Class in the Workspace Explorer, the same treeview is displayed to assist in navigation, as shown below.



The View Menu



The View menu, illustrated above, has been extended to allow you to control the new editor features, and now provides the following actions.

Trace	Displays a column to the left of the function that displays ⎕TRACE settings
Stop	Displays a column to the left of the function that displays ⎕STOP settings
Monitor	Displays a column to the left of the function that displays ⎕MONITOR settings
Line Numbers	Toggles the display of line numbers on/off.
Function Line Numbers	Toggles the display of line numbers *on individual functions* on/off. This option is only enabled when editing a Class, Namespace script or Interface.
Tree View	Toggles the display of the treeview in the left-hand pane.
Outlining	Turns outlining on and off.
Expand All Outlines	Expands all outlines.
Collapse All Outlines	Collapses all outlines
Expand all Outlines below here	Expands all outlines below the level of the current line.

Line-numbering

The manner in which the *line numbering* option works has changed.

In Version 12.1.the user's line-numbering preferences are stored in the Registry (in the Editor\Columns section) on a *per object type* basis. In other words, the system remembers the last line-numbering setting separately for a Class, function/operator, and each type of array; character vector, character matrix, vector of character vectors, and so forth.

The Line Numbers (Num-) menu item on the pop-up menu, and the [···] button on the Session toolbar (both of which are associated with the system action [LineNumbers]) toggle line-numbering on/off on the current edit window as before. However, line numbering is not changed globally but only for the *type* of object in that window.

So, for example, if you have two Edit windows open, one on a function and the other on a character matrix, toggling the line numbers on and off in one window will have no effect on the line-numbering in the other window. However, if you opened a third window on a defined operator, toggling the line-numbering in that window would affect the line-numbering in the function window.

Whenever you change line-numbering, the corresponding value in the Registry for the type of object in the currently active edit window is immediately updated.

The `lines_on_functions` parameter, which used to control line-numbering on a global basis, has been removed.

Function Line Numbers

The *Function Line Numbers* option in the pop-up Editor menu provides an additional level of line-numbering. If selected, line numbers are displayed *independently* on each individual function (or operator) in the Class. This option is only enabled when you are editing a Class, Namespace script or Interface, and is disabled for all other types of object.

Note that function line-numbering and general line-numbering are independent options and it is possible to have the entire Class numbered (from [0] to the number of lines in the Class) in addition to having line-numbering on each individual function.

Refactoring[1]

Version 12.1 provides some simple refactoring tools.

If the input cursor is positioned on a word somewhere in the body of a script, and you select one of the options from the *Refactoring* menu:

- Add text as Field
- Add text as Property
- Add text as Method

the Editor inserts the appropriate *template* for a Field, Property or Method of the same name. The template is inserted immediately above the line containing the word. So, for example, if you position the input cursor on the word `xvar`, and then choose *Add text as Property*, the Editor will insert the following lines above the line in the script containing that `xvar`:

```
:Property xvar

    ∇r←get
     r←0
    ∇

    ∇set args
    ∇
:EndProperty
```

1 *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."* -- MartinFowler inRefactoringImprovingTheDesignOfExistingCode.

Miscellaneous

The following additional improvements have been made to the Editor:

- The speed with which the Editor displays and refreshes its contents has been very substantially improved. The effect is immediate, even for very large classes, functions and variables.
- The limit on the number of lines that the Editor could effectively and correctly number, has been removed. This was previously 9999.
- If you are editing a script and you press Shift+Enter or double-click on the name of a function that is defined within the same script, the Editor will move the input cursor to that function.
- If you open the Editor on a specific name in a script, the Editor will open on the script as a whole but position the input cursor on the chosen name. For example if you enter the command:

  ```
  )ed myclass.myfn
  ```

  The Editor will open a window on `myclass` and position the input cursor on `myfn`, creating a new function if it doesn't already exist.

- When you use the Editor to view a nested array character data is displayed in the colours associated with *Character Constant* and numeric data in the colours associated with *Numeric Constant*. The same syntax colours are used to display data in the Value Tip window; the pop-up window that appears when you hover the mouse-pointer over a name.

The Tracer and Syntax Colouring

The Tracer may now optionally be coloured in the same way as the Editor. If you choose this option, the appearance of a function inside the Tracer window is almost identical to that of the Editor. The only difference is that the current line is identified by a bright red rectangle. For example:

Debugger

Tools View

gtest[4]+an←⎕NEW F

```
[0]     gtest;an;myfont
[1]     ⎕USING←'System'
[2]     ⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
[3]     ⎕USING,←⊂'System.Drawing,system.drawing.dll'
[4]     an←⎕NEW FontFamily(⊂'Arial Narrow')
[5]     myfont←⎕NEW Font(an 11 FontStyle.Bold GraphicsUnit.Point)
[6]     'f'⎕WC'Form' 'Grid using .Net TextBox Control'('Coord' 'Pixel')('Size' 130 500)
[7]     'f.g'⎕WC'Grid'('Posn' 0 0)f.Size('ShowInput' 1)('TitleWidth' 0)
[8]     wds←'All' 'TeXt' 'Is' 'Changed' 'to' 'Upper' 'casE'
[9]     wds,←'ακομα' 'kai' 'τα' 'Ελληνικα'
[10]
[11]    f.g.Values←5 5⍴wds
[12]    'f.g.tb'⎕WC'NetControl' 'TextBox'
[13]    f.g.tb.Font←myfont
[14]    f.g.tb.(CharacterCasing←CharacterCasing.Upper)
[15]    f.g.Input←'f.g.tb'
```

Modified Function | Last saved by: Pete: 13 June 2009 11:45 | Pos: 4/16,0

SIstack (Tid: 0)

If, while tracing, you decide to edit the function, the colours don't change, but the Edit toolbar replaces the Trace Tools, as shown below.

Debugger

File Edit Refactor View

```
[0]     gtest;an;myfont
[1]     ⎕USING←'System'
[2]     ⎕USING,←⊂'System.Windows.Forms,system.windows.forms.dll'
[3]     ⎕USING,←⊂'System.Drawing,system.drawing.dll'
[4]     an←⎕NEW FontFamily(⊂'Arial Narrow')
[5]     myfont←⎕NEW Font(an 11 FontStyle.Bold GraphicsUnit.Point)
[6]     'f'⎕WC'Form' 'Grid using .Net TextBox Control'('Coord' 'Pixel')('Size' 130 500)
[7]     'f.g'⎕WC'Grid'('Posn' 0 0)f.Size('ShowInput' 1)('TitleWidth' 0)
[8]     wds←'All' 'TeXt' 'Is' 'Changed' 'to' 'Upper' 'casE'
[9]     wds,←'ακομα' 'kai' 'τα' 'Ελληνικα'
[10]
[11]    f.g.Values←5 5⍴wds
[12]    'f.g.tb'⎕WC'NetControl' 'TextBox'
[13]    f.g.tb.Font←myfont
[14]    f.g.tb.(CharacterCasing←CharacterCasing.Upper)
[15]    f.g.Input←'f.g.tb'
```

Function | Last saved by: Pete: 13 June 2009 11:45 | Pos: 2/16,0

SIstack (Tid: 0)

The option to apply syntax colouring to the Tracer is set by checking the *Function Tracer* button on the *Syntax* tab of the *Colour Selection* dialog box. If you check this option, the Tracer colour scheme defined from the *Session/Trace* tab is overridden.

APLFormatBias Parameter

In Version 11.0, ⎕FMT and dyadic ⍕ were changed to improve the accuracy of the display of floating-point numbers. The gain in accuracy has necessarily been achieved at the cost of a reduction in performance. The difference in performance is significant, but may only be noticeable in an application that does a lot of numeric formatting.

In versions prior to Version 11.0, there was a significant performance optimisation whereby numbers which would display in fewer than 10 digits were scaled and then treated as integers.

A second consequence of the optimisation was that numbers smaller than but indistinguishable (in terms of the accuracy of internal representation) from n.5 were rounded up. In other words, any number whose decimal part (the part of the number after the decimal point) is in the range (0.49999999999999987 to 0.49999999999999999 will be rounded as if it were 0.5 (i.e., rounded up). This effect, although potentially desirable, may be regarded as being not strictly correct.

For example, consider the number 0.015 which is internally represented as the value 1.4999999...

```
      2 ⍕0.015  ⍝ Versions prior to 11.0
 0.02

      2 ⍕0.015  ⍝ Version 11.0 onwards
 0.01
```

A new parameter `APLFormatBias` has been introduced (in Version 12.0 Release 4) to allow you to select the old, pre-Version 11.0 behaviour.

If you wish to get the old pre-Version 11.0 behaviour, which includes significantly better performance, you should set `APLFormatBias` to 1. The default is 0 (slower but mathematically more “correct”). As with any other parameter, you may set `APLFormatBias` in the Windows Registry, on the command line, or as an environment variable.

Miscellaneous

Change to Outer-Product

In Version 12.1, the way that outer-product produces a null result has changed.

Outer-product now applies its operand function once between the first items of its arguments to determine the prototype of the result. In previous versions, the function was not applied, and the prototype of the result was the same as that of the right argument. This lead to anomalies such as `∘.≡` producing a (null) *character* result.

Change to ⎕EX and ⎕SIZE

`⎕EX` and `⎕SIZE` now accept a vector of character vectors as their argument..

Change to ⎕AVU

In version 12.1, there is a small change to the default ⎕AVU as follows:

```
⎕AVU[⎕IO+219 237]←180 96
```

Previously, these two elements had the value 8217 8218.

The Unicode code points involved are:

0096: U+0060 GRAVE ACCENT
0180: U+00B4 ACUTE ACCENT
8216: U+2018 LEFT SINGLE QUOTATION MARK
8217: U+2019 RIGHT SINGLE QUOTATION MARK

Change to SALT

The default target space for the SALT “Load” command has changed. Previously, the default was always to load objects into the root namespace (#). The version of SALT distributed with Version 12.1 will load into the space from which the call to SALT was made. You can achieve the old behaviour by adding the the switch ‘-target=#’ to explicitly set the target to #.

System Requirements

Microsoft Windows

Dyalog APL Version 12.1 supports the following Versions of Windows:

- Windows Vista
- Windows XP
- Windows 2000
- Windows Server 2003

Note that Dyalog APL Version 12.1 is not supported under Windows 95, Windows 98, Windows ME or Windows NT4.

Microsoft .Net Interface

Dyalog APL Version 12.1 .Net Interface requires Version 2.x or greater of the Microsoft .Net Framework. It does *not* operate with .Net Version 1.0.

Interoperability and Compatibility

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example a file component written by a PC will almost certainly have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 12.1 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. From Version 11.0, component files and workspaces can generally be shared between Dyalog interpreters running on different platforms. However, this is not always possible. For example, component files created by Version 10.1 can often not be shared across platforms, even when used by later versions (the system function ⎕FCOPY can be used to make a logically identical copy of an old file, which is fully inter-operable).

The following sections describe other limitations in inter-operability:

Code

Code which is saved in workspaces, or embedded within ⎕ORs stored in component files, can generally only be read by the version which saved them and later versions of the interpreter. In the case of workspaces, a load (or copy) from an older version would fail with the message:

```
      this WS requires a later version of the interpreter.
```

In the case of ⎕OR, unpredictable behaviour may result if an older version reads a ⎕OR saved by a later version of the system. Thus, ⎕OR is ***not recommended*** as a mechanism for sharing code or objects between different versions of APL.

“Ordinary” Arrays

With the exception of the Unicode restrictions described in the following paragraphs, Dyalog APL provides complete inter-operability for arrays which only contain (nested) character and numeric data. Such arrays can be stored in component files - or transmitted using TCPSocket objects and Conga connections, and shared between all versions and across all platforms.

As mentioned in the introduction, full cross-platform interoperability of component files is only available for large component files (see the following section), and for small component files created by Version 11.0 or later.

32 vs 64-bit Component Files

Large (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction (versions earlier than 10.1).

The second item in the right argument of ⎕FCREATE determines the addressing type of the file.

```
      'small'⎕fcreate 1 32    ⍝ create small file.
      'large'⎕fcreate 1 64    ⍝ create large file.
```

If the second item is missing, the file type defaults to 64-bit-addressing. In versions prior to 12.0, the default was 32-bit addressing.

Note that *small* (32-bit-addressing) cannot contain Unicode data. Unicode editions of Dyalog APL can only write character data which would be readable by a Classic edition (consisting of elements of ⎕AV).

External Variables

External variables are implemented as small (32-bit -adressing) component files, and subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please contact Dyalog if you need further advice on this topic.

32 vs. 64-bit Interpreters

From Dyalog APL Version 11.0 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures (in general, the 32-bit versions will also run on 64-bit machines running 64-bit operating systems). There is complete inter-operability between 32- and 64-bit interpreters.

Unicode vs. Classic Editions

From Version 12.0 onwards, a Unicode edition is available, which is able to work with the entire Unicode character set. Classic editions (a term which includes versions prior to 12.0) are limited to the 256 characters defined in the atomic vector, ⎕AV). Large (64-bit-addressing) component files have a Unicode property; when this is enabled, Unicode data may be stored in the file. The Unicode property is always off for small (32-bit addressing) files, which may not contain Unicode data. When a Unicode edition writes to a component file which may not contain Unicode data, character data is mapped to ⎕AV, and can therefore be read without problems by Classic editions.

A TRANSLATION ERROR will occur if a Unicode edition writes to a non-Unicode component file, if the data being written contains characters which are not in ⎕AV (see ⎕AVU for more details). Likewise, a Classic edition (Version 12.0 or later) will issue a TRANSLATION ERROR if it reads Unicode data from a component file, and is unable to map it to ⎕AV. Version 10.1 cannot read components containing Unicode data.

A TRANSLATION ERROR will also issued when a Classic edition)LOADs or)COPYs a workspace containing Unicode data which cannot be mapped to ⎕AV.

TCPSocket objects have an APL property which corresponds to the Unicode property of a file, if this is set to Classic (the default) the data in the socket will be restricted to ⎕AV, if Unicode it will contain Unicode character data. As a result, TRANSLATION ERRORs can occur on transmission or reception in the same way as when updating or reading a file component.

File Journaling

Version 12.0 introduces File Journaling (level 1), and 12.1 adds levels 2 and 3. Versions earlier than 12.0 cannot tie files which have any form of journaling enabled. Version 12.0 cannot tie files with journaling levels other than 1. Files can be shared with earlier versions by using ⎕FPROPS to switch journaling off.

Auxiliary Processors

A Dyalog APL process is restricted to starting an AP of exactly the same architecture. In other words, the AP must share the same word-width and byte-ordering as its interpreter process.

Session Files

Session (.dse) files may only be used on the platform on which they were created and saved.

CHAPTER 2

XML Import and Export

Introduction

The ⎕XML system function allows XML (eXtensible Markup Language) format data to be efficiently converted to and from arrays which have a structure designed to be easy for APL programmers to generate and manipulate. This allows APL applications to easily import and export data in XML format, to exchange data with other applications written in APL (including other dialects of APL), or other languages.

Example 1 - converting XML text to a matrix:

```
      x←'<xml><document id="001">An introduction to XML'
      x,←'</document></xml>'
      x
<xml><document id="001">An introduction to XML</document>
</xml>

      v←⎕xml x

      ]display ⎕XML x
┌→──────────────────────────────────────────────────────────────────────┐
↓   ┌→──┐    ┌⊖┐                               ┌→──────────────┐      │
│ 1 │xml│    │ │                               ⌽ ┌⊖┐  ┌⊖┐      │    3 │
│   └───┘    └─┘                               │ │ │  │ │      │      │
│                                              │ └─┘  └─┘      │      │
│                                              └∊──────────────┘      │
│   ┌→───────┐ ┌→────────────────────┐         ┌→──────────────────┐  │
│ 2 │document│ │An introduction to XML│        ↓ ┌→─┐  ┌→──┐       │1 │
│   └────────┘ └─────────────────────┘         │ │id│  │001│       │  │
│                                              │ └──┘  └───┘       │  │
│                                              └∊──────────────────┘  │
└∊──────────────────────────────────────────────────────────────────────┘
```

Example 2 – converting a matrix to XML text

```
      ⎕XML v
<xml>
  <document id="001">An introduction to XML
  </document>
</xml>
```

Principle features:

- All well-formed XML is accepted by ⎕XML and converted into an APL array for use in an APL application.
- XML is generated by ⎕XML from a matrix in the correct format,
- Data and data format described by XML is stored in the APL array in a format which is designed to be easy to generate and manipulate from an APL program.
- XML grammar such as processing instructions, document type declarations etc may *optionally* be stored in the APL array, but will not be processed or validated. This is principally to allow regeneration of XML from XML input which contains such structures, but an APL application could process the data if it chose to do so.
- XML is an open standard, designed to allow exchange of data between applications. The full specification (http://www.w3.org/TR/2008/REC-xml-20081126/) describes functionality, including processing directives and other directives, which can transform XML data as it is read, and which a full *XML processor* would be expected to handle. The ⎕XML function is designed to handle XML to the extent required to import and export APL data. It favours speed over complexity - some markup is tolerated but largely ignored, and there are no XML query or validation features. APL applications which require processing, querying or validation will need to call external tools for this, and finally call ⎕XML on the resulting XML to perform the transformation into APL arrays.

XML Processing

The XML definition uses specific terminology to describe its component parts. This is a summary of the terms used in this chapter:

Character Data

Character data consists of free-form text. The free-form text should not include the characters '>' (when it follows ']]'), '<' or '&', so these must be represented by their entity references ('>', '<' and '&' respectively), or numeric character references.

Entity References and Character References

Entity references are named representations of single characters which cannot normally be used in character data because they are used to delimit markup, such as > for '>'. Character references are numeric representations of any character, such as for space. Note that character references always take values in the Unicode code space, regardless of the encoding of the XML text itself.

⎕XML converts entity references and all character references which the APL character set is able to represent into their character equivalent when generating APL array data; when generating XML it converts any or all characters to entity references as needed.

There is a predefined set of entity references, and the XML specification allows others to be defined within the XML using the <!ENTITY > markup. ⎕XML does not process these additional declarations and therefore will only convert the predefined types.

Whitespace

Whitespace sequences consist of one or more spaces, tabs or line-endings. Within character data, sequences of one or more whitespace characters are replaced with a single space when this is enabled by the whitespace option. Line endings are represented differently on different systems (0x0D 0x0A, 0x0A and 0x0D are all used) but are normalized by converting them all to 0x0A before the XML is parsed, regardless of the setting of the whitespace option.

Elements

An element consists of a balanced pair of tags or a single empty element tag. Tags are given names, and start and end tag names must match.

An example pair of tags, named TagName is

```
<TagName></TagName>
```

This pair is shown with no content between the tags; this may be abbreviated as an empty element tag as

```
<TagName/>
```

Tags may be given zero or more attributes, which are specified as name/value pairs; for example

```
<TagName AttName="AttValue">
```

Attribute values may be delimited by either double quotes as shown or single quotes (apostrophes); they may not contain certain characters (the delimiting quote, '&' or '<') and these must be represented by entity or character references.

The content of elements may be zero or more mixed occurrences of character data and nested elements. Tags and attribute names *describe* data, attribute values and the content within tags contain the data itself. Nesting of elements allows structure to be defined.

Because certain markup which describes the format of allowable data (such as element type declarations and attribute-list declarations) is not processed, no error will be reported if element contents and attributes do not conform to their restricted declarations, nor are attributes automatically added to tags if not explicitly given.

Attributes with names beginning ‘xml:’ are reserved. Only **xml:space** is treated specially by ⎕XML. When converting both from and to XML, the value for this attribute has the following effects on space normalization for the character data within this element and child elements within it (unless subsequently overridden):

- **default** – space normalization is as determined by the **whitespace** option.
- **preserve** - space normalization is disabled – all whitespace is preserved as given.
- **any other value** – rejected.

Regardless of whether the attribute name and value have a recognised meaning, the attribute will be included in the APL array / generated XML. Note that when the names and values of attributes are examined, the comparisons are case-sensitive and take place after entity references and character references have been expanded.

Comments

Comments are fully supported markup. They are delimited by ‘<!--‘ and ‘-->’ and all text between these delimiters is ignored. This text is included in the APL array if markup is being preserved, or discarded otherwise.

CDATA Sections

CDATA Sections are fully supported markup. They are used to delimit text within character data which has, or may have, markup text in it which is not to be processed as such. They and are delimited by ‘<![CDATA[‘ and ‘]]>’. CDATA sections are never recorded in the APL array as markup when XML is processed – instead, that data appears as character data. Note that this means that if you convert XML to an APL array and then convert this back to XML, CDATA sections will not be regenerated. It is, however, *possible* to generate CDATA sections in XML by presenting them as markup.

Processing Instructions

Processing Instructions are delimited by ‘<&’ and ‘&>’ but are otherwise treated as other markup, below.

Other markup

The remainder of XML markup, including document type declarations, XML declarations and text declarations are all delimited by ‘<!’ and ‘>’, and may contain nested markup. If markup is being preserved the text, including nested markup, will appear as a single row in the APL array. ⎕XML does not process the contents of such markup. This has varying effects, including but not limited to the following:

- No validation is performed.
- Constraints specified in markup such element type declarations will be ignored and therefore syntactically correct elements which fall outside their constraint will not be rejected.
- Default attributes in attribute-list declarations will not be automatically added to elements.
- Conditional sections will always be ignored.
- Only standard, predefined, entity references will be recognized; entity declarations which define others entity references will have no effect.
- External entities are not processed.

Conversion from XML

When the right argument to ⎕XML is a character vector it is assumed to contain XML text. This is processed to generate a five column array with the following information in the columns:

1. Numeric value which indicates the nesting depth.
2. Element name, other markup text, or empty character vector when empty.
3. Character data, or empty character vector when empty.
4. Attribute name and value pairs, `0 2⍴⊂''` when empty.
5. A numeric value which indicates what the row contains.

Notes:

- The level number in the first column is 0 for the outermost level and subsequent levels are represented by an increase of 1 for each level. Thus, for

  ```
  <xml><document id="001">An introduction to XML
  </document></xml>
  ```

  The `xml` element is at level 0 and the `document id` element is at level 1. The text within the `document id` element is at level 2.

- Each tag in the XML contains an element name and zero or more attribute name and value pairs, delimited by ‘<’ and ‘>’ characters. The delimiters are not included in the result matrix. The element name of a tag is stored in column 2 and the attribute(s) in column 4.

- All XML markup other than tags are delimited by either ‘<!’ and ‘>’, or ‘<?’ and ‘>’ characters. By default these are not stored in the result matrix but the **markup** option may be used to specify that they are. The elements are stored in their entirety, except for the leading and trailing ‘<’ and ‘>’ characters, in column 2. Nested constructs are treated as a single block. Because the leading and trailing ‘<’ and ‘>’ characters are stripped, such entries will always have either ‘!’ or ‘&’ as the first character.

- Character data itself has no tag name or attributes. As an optimization, when character data is the sole content of an element, it is included with its parent rather than as a separate row in the result. Note that when this happens, the level number stored is that of the parent; the data itself implicitly has a level number one greater.

- Attribute name and value pairs associated with the element name are stored in the fourth column, in an (*n x 2*) matrix of character values, for the *n* (including zero) pairs.

- Each row is further described in the fifth column as a convenience to simplify processing of the array (although this information could be deduced). The following values are defined:

1	Element
2	Child element
4.	Character data
8.	Markup not otherwise defined
16.	Comment markup
32.	Processing instruction markup

- Any given row may contain an entry for an element, character data, markup not otherwise defined, a comment or a processing instruction. Furthermore, an element will have zero or more of these as children. For all types except elements, the value in the fifth column is as shown above. For elements, the value is computed by adding together the value of the row itself (1) and those of its children. For example, the value for a row for an element which contains one or more sub-elements and character data is 7 – that is 1 (element) + 2 (child element) + 4 (character data). It should be noted that:

    1. Odd values always represent elements. Odd values other than 1 indicate that there are children.
    2. Elements which contain just character data (5) are combined into a single row as noted previously.
    3. Only immediate children are considered when computing the value. For example, an element which contains a sub-element which in turn contains character data does not itself contain character data.
    4. The computed value is derived from what is actually preserved in the array. For example, if the source XML contains an element which contains a comment, but comments are being discarded, there will be no entry for the comment in the array and the fifth column for the element will not indicate that it has a child comment.

Conversion to XML

Conversion to XML takes an array with the format described above and generates XML text from it. There are some simplifications to the array which are accepted:

- The fifth column is not needed for XML generation and is effectively ignored. Any numeric values are accepted, or the column may be omitted altogether.
- If there are no attributes in a particular row then the `0 2⍴⊂''` may be abbreviated as zilde. If the fifth column is omitted then the fourth column may also be omitted altogether.
- Data in the third column and attribute values in the fourth column (if present) may be provided as either character vectors or numeric values. Numeric values are implicitly formatted as if ⎕PP was set to 17.

The following validations are performed on the data in the array:

- All elements within the array are checked for type.
- Values in column 1 must be non-negative and start from level 0, and the increment from one row to the next must be ≤ +1.
- Tag names in column 2 and attribute names in column 4 (if present) must conform to the XML name definition.

Then, character references and entity references are emitted in place of characters where necessary, to ensure that valid XML is generated. However, markup, if present, is *not* validated and it is possible to generate invalid XML if care in not taken with markup constructs.

CHAPTER 3

Language Enhancements

New and Revised Primitive & System Functions

New Primitive Operators, Functions, System Functions & Variables

⍪	Table
⌶	I-Beam
⎕FCHK	File Check and Repair
⎕XML	XML Convert

Revised Primitive Functions, System Functions & Variables

∘.	Outer Product
⎕EX	Expunge Object(s)
⎕FPROPS	File Properties
⎕SIZE	Size of Object(s)

Table: R←⍪Y

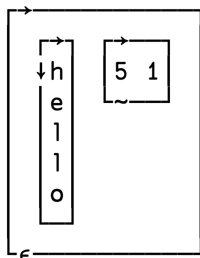
Y may be any array. R is a 2-dimensional matrix of the elements of Y taken in row-major order, preserving the shape of the first dimension of Y if it exists

Table has been implemented according to the Extended APL Standard (ISO/IEC 13751:2001).

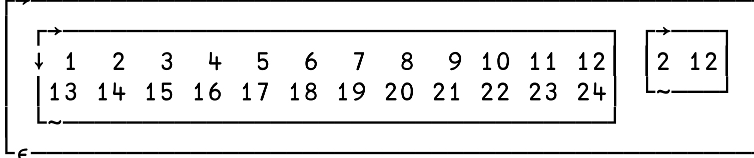
Examples

```
      ]display {⍵ (⍴⍵)} ⍪'a'
┌→────────────┐
│ ┌→┐ ┌→──┐   │
│ ↓a│ │1 1│   │
│ └─┘ └~──┘   │
└∊────────────┘
```

```
      ]display {⍵ (⍴⍵)} ⍪'hello'
```



```
      ]display {⍵ (⍴⍵)} ⍪2 3 4⍴⍳24
```



I-Beam: R←{X}(A⌶)Y

I-Beam is a monadic operator that provides a range of system related services.

WARNING: Although documentation is provided for I-Beam functions, any service provided using I-Beam should be considered as "experimental" and subject to change – without notice - from one release to the next. Any use of I-Beams in applications should therefore be carefully isolated in cover-functions that can be adjusted if necessary.

A is an integer that specifies the type of operation to be performed as shown in the table below. Y is an array that supplies further information about what is to be done.

X is currently unused.

R is the result of the derived function.

A	**Derived Function**
200	Syntax Colouring
1111	Number of Threads
1112	Parallel Execution Threshold
1113	Thread Synchronisation Mechanism
2000	Memory Manager Statistics
2110	Export to Memory

Syntax Colouring: R←200⌶Y

This function obtains syntax colouring information for a function.

Y is a vector of character vectors containing the ⎕NR representation of a function or operator.

R is a vector of integer vectors with the same shape and structure of Y in which each number identifies the syntax colour element associated with the corresponding character in Y.

```
      {(↑ω),↑ 200⌶ω}  'foo; local' 'global'
'local←ρρ''hello'''

foo; local      21 21 21 19  3 31 31 31 31 31 0 0 0 0 0
global           7  7  7  7  7  7  0  0  0  0 0 0 0 0 0
local←ρρ'hello' 31 31 31 31 31 19 23 23  4  4 4 4 4 4 4
```

In this example:
21 is the syntax identifier for “function name”
19 is the syntax identifier for “primitive”
3 is the syntax identifier for “white space”
31 is the syntax identifier for “local name”
7 is the syntax identifier for “global name”
23 is the syntax identifier for “idiom”

Number of Threads: R←1111⌶Y

Specifies how many threads are to be used for parallel execution.

Y is an integer that specifies the number of threads that are to be used henceforth for parallel execution. Prior to this call, the default number of threads is specified by an environment variable named APL_MAX_THREADS. If this variable is not set, the default is the number of CPUs that the machine is configured to have.

R is the previous value

Note that (unless APL_MAX_THREADS is set), the number of CPUs for which the machine is configured is returned by the first execution of 1111⌶. The following expression obtains and resets the number of threads back to this value.

```
      {}1111⌶ ncpu←1111⌶1
```

Parallel Execution Threshold: R←1112⌶Y

Y is an integer that specifies the array size threshold at which parallel execution takes place. If a parallel-enabled function is invoked on an array whose number of elements is equal to or greater than this threshold, execution takes place in parallel. If not, it doesn’t.

Prior to this call, the default value of the threshold is specified by an environment variable named APL_MIN_PARALLEL. If this variable is not set, the default is 32768.

R is the previous value

Thread Synchronisation Mechanism: R←1113⌶Y

Y is Boolean and specifies whether or not the main thread does a busy wait for the others to complete or uses a semaphore when a function is executed in parallel.

The default and recommended value is 0 (use a semaphore). This function is provided only for Operating Systems that do not support semaphores.

A value of 1 **must** be set if you are running AIX Version 5.2 which does not support Posix semaphores. Later versions of AIX do not have this restriction.

R is the previous value

Memory Manager Statistics: R←2000⌶Y

This function returns information about the state of the workspace. This I-Beam is provided for performance tuning and is VERY LIKELY to change in the next release.

Y is a simple scalar or vector containing up to 5 elements.

The result R is an array with the same structure as Y, but any elements of Y that contain values other than ¯1, are replaced by the following values according to their position.

Index	Description
1	Workspace available (a "quick" ⎕WA)
2	Workspace used
3	Number of compactions since the workspace was loaded
4	Number of garbage collections that found garbage
5	Current number of garbage pockets in the workspace

Note that while the first 4 operations are relatively fast, the fifth operation may take a noticeable amount of time, depending upon the size and state of the workspace.

Examples

```
      2000⌶0
65374272

      2000⌶5⍴0
65374272 184256 2 1 0

      2000⌶¯1 ¯1 0 0
¯1 ¯1 2 1
```

Export to Memory: R←2100⌶Y

This function exports the current active workspace as an in-memory .NET.Assembly.

Y may be any array and is ignored.

The result R is 1 if the operation succeeded or 0 if it failed.

Expunge Object: {R}←⎕EX Y

Y must be a simple character scalar, vector or matrix, or a vector of character vectors containing a list of names. R is a simple Boolean vector with one element per name in Y.

Each name in Y is disassociated from its value if the active referent for the name is a defined function, operator, variable or namespace.

The value of an element of R is 1 if the corresponding name in Y is now available for use. This does not necessarily mean that the existing value was erased for that name. A value of 0 is returned for an ill-formed name or for a distinguished name in Y. The result is suppressed if not used or assigned.

Examples

```
      ⎕EX'VAR'
      +⎕EX'FOO' '⎕IO' 'X' '123'
1 0 1 0
```

If a named object is being executed the existing value will continue to be used until its execution is completed. However, the name becomes available immediately for other use.

Examples

```
      )SI
FOO[1]*

      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇
      +⎕EX'FOO'
1
      )SI
FOO[1]*


     ∇FOO[⎕]
defn error

      FOO←1 2 3
      →⎕LC
10
      FOO
1 2 3
```

If a named object is an external variable, the external array is disassociated from the name:

```
      ⎕XT'F'
FILES/COSTS
      ⎕EX'F' ⋄ ⎕XT'F'
```

If the named object is a GUI object, the object and all its children are deleted and removed from the screen. The expression `⎕EX'.'` deletes all objects owned by the current thread **except** for the Root object itself. In addition, if this expression is executed by thread 0, it resets all the properties of `'.'` to their default values. Furthermore, any unprocessed events in the event queue are discarded.

If the named object is a shared variable, the variable is retracted.

If the named object is the last remaining external function of an auxiliary process, the AP is terminated.

If the named object is the last reference into a dynamic link library, the DLL is freed.

File Check and Repair: R←{X} ⎕FCHK Y

⎕FCHK validates and repairs component files following an abnormal termination of the APL process or operating system.

Y must be a simple character scalar or vector which specifies the name of the file to be exclusively checked or repaired. The file must be named in accordance with the operating system's conventions, and may be a relative or absolute pathname. The file must exist and must not already be tied.

The optional left-argument X must be a vector of zero or more character vectors from among 'force', 'repair' and 'rebuild', which determine the detailed operation of the function. Note that these options are case-sensitive.

- If X contains 'force' ⎕FCHK will validate the file even if it appears to have been cleanly untied.
- If X contains 'repair' ⎕FCHK will repair the file, following validation, if it appears to be damaged. This option may be used in conjunction with 'force'.
- If X contains 'rebuild' ⎕FCHK will repair the file unconditionally.

If X is omitted, the default behaviour is as follows:

1. If the file appears to have been cleanly untied previously, return ⍬, i.e. report that the file is OK.
2. Otherwise, validate the file and return the appropriate result. If the file is corrupt, no attempt is made to repair it.

The result R is a vector of the numbers of missing or damaged components. R may include non-positive numbers of "pseudo components" that indicate damage to parts of the file other than in specific components:

0	ACCESS MATRIX.
¯1	Free-block tree.
¯2	Component index tree.

Other negative numbers represent damage to the file metadata; this set may be extended in the future.

Following a *check* of the file, a non-null result indicates that the file is damaged.

Following a *repair* of the file, the result indicates those components that could not be recovered. Un-recovered components will give a FILE COMPONENT DAMAGED error if read but may be replaced without error.

Repair can recover only check-summed components from the file, i.e. only those components that were written with the checksum option enabled (see ⎕FPROPS).

Following an operating system crash, repair may result in one or more individual components being rolled back to a previous version or not recovered at all, unless Journaling levels 2 or 3 were also set when these components were written.

File Properties: R←X ⎕FPROPS Y

Access Code 1 (to read) or 8192 (to change properties)

⎕FPROPS reports and sets the properties of a component file.

Y must be a simple integer scalar or vector containing the file tie number.

X must be a simple character scalar or vector containing one or more valid Identifiers listed in the table below, or a vector of 2-element vectors, each of which contains an Identifier and a (new) value for that property.

If the left argument is a simple character array, the result R contains the current values for the properties identified by X. If the left argument is nested, the result R contains the previous values for the properties identified by X.

Identifier	Property	Description / Legal Values
S	File Size (read only)	32 = Small Component Files (<4Gb) 64 = Large Component Files
E	Endian-ness (read only)	0 = Little-endian 1 = Big-endian
U	Unicode	0 = Characters must be written as type 82 arrays 1 = Characters must be written as Unicode arrays
J	Journaling	0 = Disable Journaling 1 = Enable *APL crash proof* Journaling 2 = Enable *System crash proof* Journaling; repair needed on recovery 3 = Enable full *System crash proof* Journaling
C	Checksum	0 = Disable checksum 1 = Enable checksum

The default properties for a newly created file are as follows:

- S = 64
- U = 1 (Unicode Edition and 64-bit file) or 0 (otherwise)

- J = 0
- C = 0
- E depends upon the computer architecture.

Journaling Levels

Level 1 journaling (APL crash-proof) automatically protects a component file from damage in the event of abnormal termination of the APL process. The file state will be implicitly committed between updates and an incomplete update will automatically be rolled forward or back when the file is re-tied. In the event of an operating system crash the file may be more seriously damaged. If checksum was also enabled it may be repaired using ⎕FCHK but some components may be restored to a previous state or not restored at all.

Level 2 journaling (system crash-proof – repair needed on recovery) extends level 1 by ensuring that a component file is fully repairable using ⎕FCHK with no component loss in the event of an operating system failure. If an update was in progress when the system crashed the affected component will be rolled back to the previous state. Tying and modifying such a file without first running ⎕FCHK may however render it un-repairable.

Level 3 journaling (system crash-proof) extends level 1 further by protecting a component file from damage in the event of abnormal termination of the APL process and also the operating system. Rollback of an incomplete update will be automatic and no explicit repair will be needed.

Enabling journaling on a component file will reduce performance of file updates; higher journaling levels have a greater impact.

Journaling levels 2 and 3 cannot be set unless the checksum option is also enabled.

Checksum Option

Enabling the checksum option for a component file will enable a damaged file to be repaired using ⎕FCHK. It will however will reduce the performance of file updates slightly and result in larger component files. The use of Check Sums is highly recommended, but in order to provide compatibility with earlier versions of Dyalog, it is not enabled by default. When version 12.1 becomes the oldest supported version, Check Sums will probably be enabled by default.

Enabling the checksum option on an existing non-empty component file, will mean that all components that had previously been written without a checksum, will be check-summed and converted. This operation which will take place when ⎕FPROPS is changed, may not, therefore, be instantaneous.

Journaling and checksum settings may be changed at any time a file is exclusively tied.

Component files written with Checksum enabled cannot be read by versions of Dyalog APL prior to Version 12.1.

Example

```
      tn←'myfile64' ⎕FCREATE 0
      'SEUJ' ⎕FPROPS tn
64 0 1 0

      tn←'myfile32' ⎕FCREATE 0 32
      'SEUJ' ⎕FPROPS tn
32 0 0 0
```

The following expression disables Unicode and switches Journaling on. The function returns the previous settings:

```
      ('U' 0)('J' 1) ⎕FPROPS tn
1 0
```

The Unicode property applies only to 64-bit component files. 32-bit component files *may not* contain Unicode character data and the value of the Unicode property is always 0. To convert a 32-bit component file to a 64-bit component file, use `⎕FCOPY`.

Properties may be read by a task with `⎕FREAD` permission (access code 1), and set by a task with `⎕FSTAC` access (8192). To set the value of the Journaling property, the file must be exclusively tied.

If Journaling or Unicode properties are set, the file cannot be tied by Versions prior to Version 12.0. If journaling is set to a value higher than 1, or checksums are enabled, the file cannot be tied by versions prior to 12.1.

Size of Object: R←⎕SIZE Y

Y must be a simple character scalar, vector or matrix, or a vector of character vectors containing a list of names. R is a simple integer vector of non-negative elements with the same length as the number of names in Y.

If the name in Y identifies an object with an active referent, the workspace required in bytes by that object is returned in the corresponding element of R. Otherwise, 0 is returned in that element of R.

The result returned for an external variable is the space required to store the external array. The result for a system constant, variable or function is 0. The result returned for a GUI object gives the amount of workspace needed to store it, but excludes the space required for its children.

Note: Wherever possible, Dyalog APL *shares* the whole or part of a workspace object rather than generates a separate copy.

Examples

```
      ⎕VR 'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      A←⍳10

      'EXT/ARRAY' ⎕XT'E' ⋄ E←⍳20

      ⎕SIZE 'A' 'FOO' 'E' 'UND'
28 76 120 0
```

XML Convert: R←{X} ⎕XML Y

⎕XML converts a string containing XML into an APL array, and "exports" an APL array as an XML string.

For conversion *from* XML, Y is a character vector containing an XML string. The result R is a 5 column matrix.

For conversion *to* XML, Y is a 3, 4 or 5 column matrix and the result R is a character vector.

The optional left argument X specifies a set of option/value pairs, each of which is a character vector. X may be a 2-element vector, or a vector of 2-element character vectors.

There are 3 option names; whitespace , markup, and unknown-entity whose possible values are summarised below. Note that the **default** value is shown in bold text, and that the option names and values are case-sensitive.

Errors detected in the input arrays or options will all cause DOMAIN ERROR.

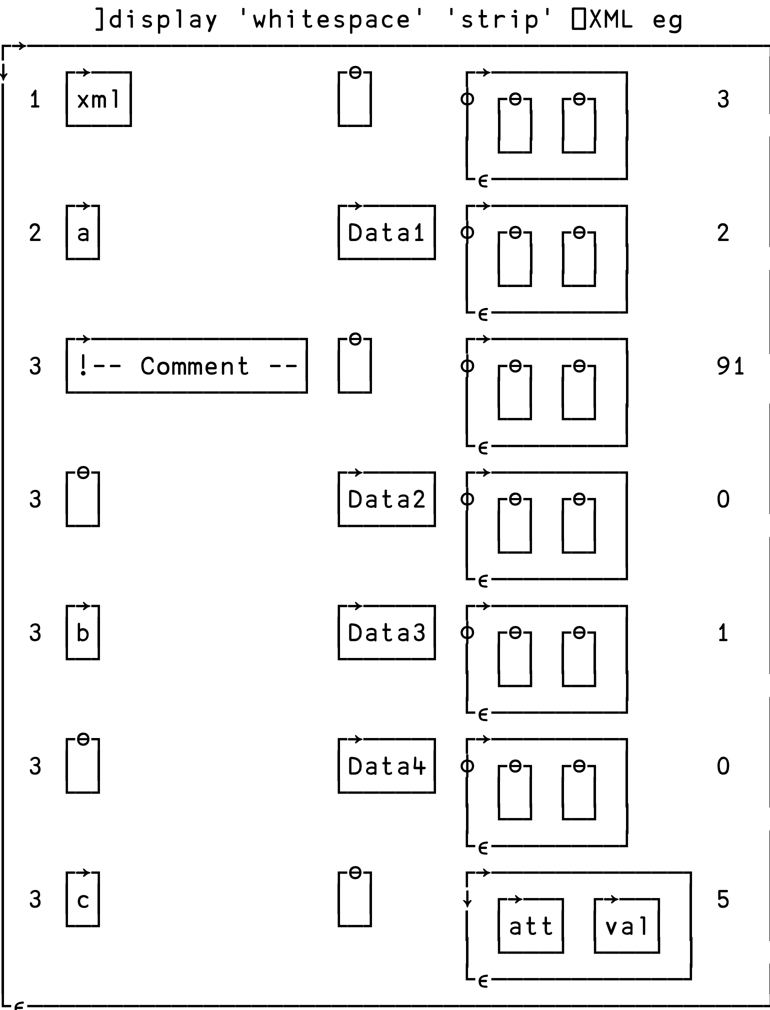
whitespace

When converting from XML `whitespace` specifies the default handling of white space surrounding and within character data. When converting to XML `whitespace` specifies the default formatting of the XML. Note that attribute values are not comprised of character data so whitespace in attribute values is always preserved.

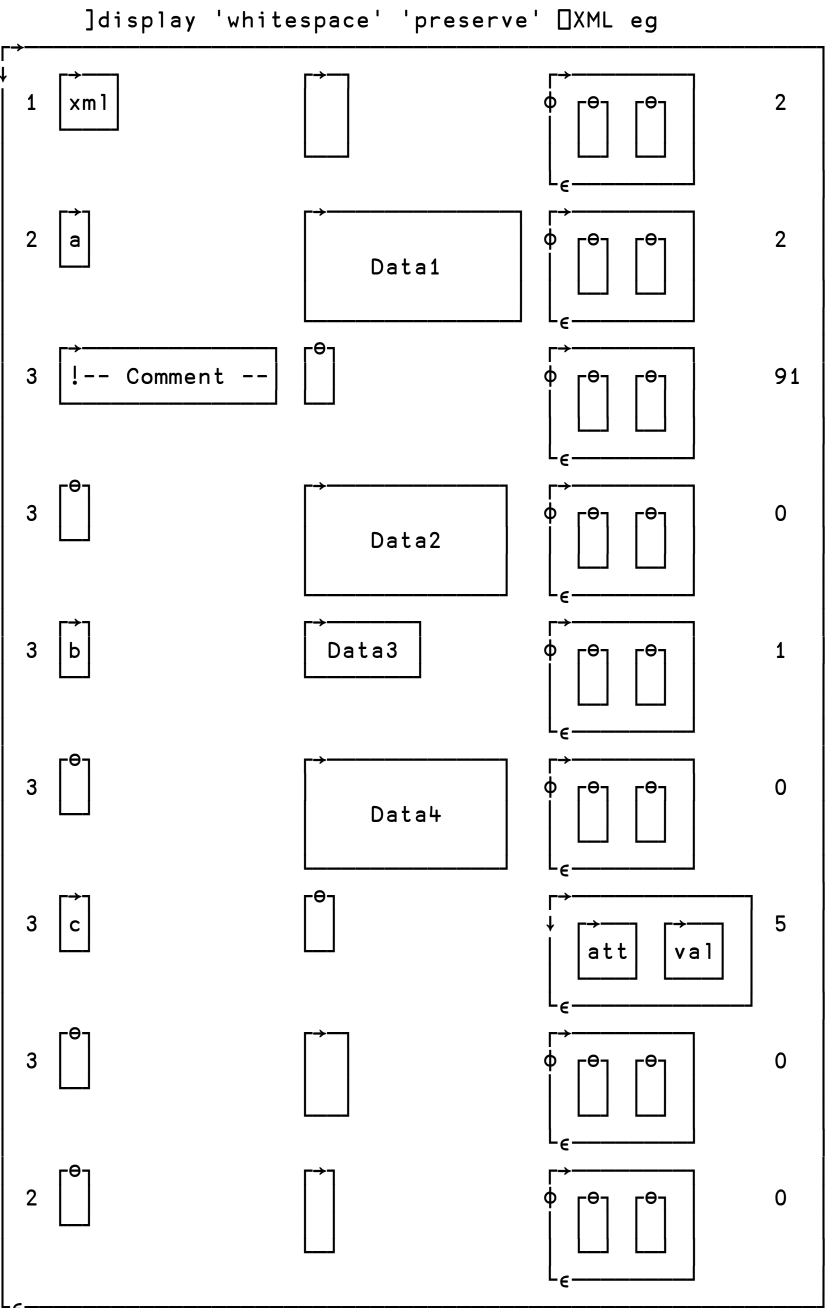
Converting from XML	
`strip`	All leading and trailing whitespace sequences are removed; remaining whitespace sequences are replaced by a single space character.
`trim`	All leading and trailing whitespace sequences are removed; all remaining whitespace sequences are handled as `preserve`.
`preserve`	Whitespace is preserved as given except that line endings are represented by Linefeed (`⎕UCS 10`).
Converting to XML	
`strip`	All leading and trailing whitespace sequences are removed; remaining whitespace sequences within the data are replaced by a single space character. XML is generated with formatting and indentation to show the data structure.
`trim`	Synonymous with `strip`.
`preserve`	Whitespace in the data is preserved as given, except that line endings are represented by Linefeed (`⎕UCS 10`). XML is generated with no formatting and indentation other than that which is contained within the data.

```
      ]display eg
┌→─────────────────────┐
│<xml>                 │
│  <a>                 │
│    Data1             │
│    <!-- Comment -->  │
│    Data2             │
│    <b> Data3 </b>    │
│    Data4             │
│    <c att="val"/>    │
│  </a>                │
│</xml>                │
└──────────────────────┘
```

```
      ]display 'whitespace' 'strip' ⎕XML eg
```



```
      ]display 'whitespace' 'preserve' ⎕XML eg
```



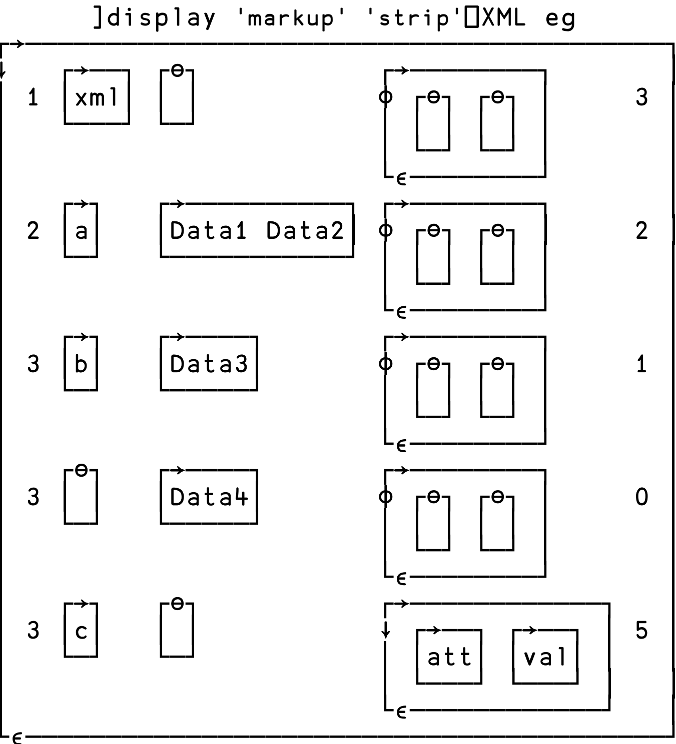
markup

When converting from XML, `markup` determines whether markup (other than entity tags) appears in the output array or not. When converting to XML `markup` has no effect.

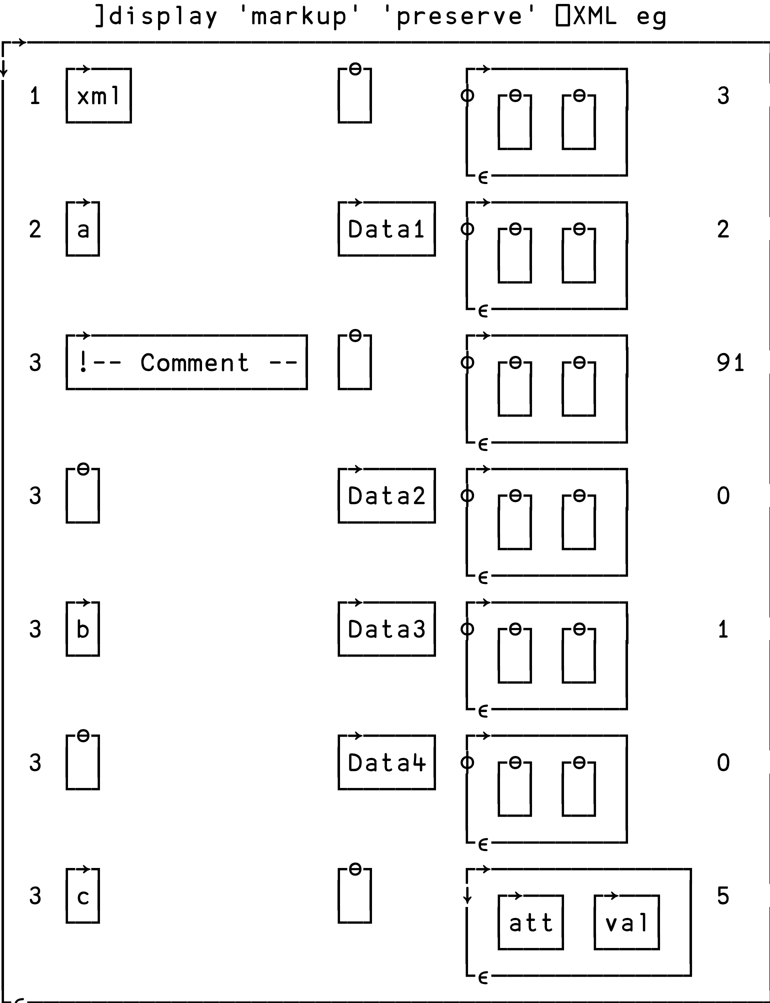
Converting from XML	
`strip`	Markup data is not included in the output array.
`preserve`	Markup text appears in the output array, without the leading '<' and trailing '>' in the tag (2nd) column.

```
      ]display eg
┌→──────────────────────┐
│<xml>                  │
│  <a>                  │
│    Data1              │
│    <!-- Comment -->   │
│    Data2              │
│    <b> Data3 </b>     │
│    Data4              │
│    <c att="val"/>     │
│  </a>                 │
│</xml>                 │
└───────────────────────┘
```

```
      ]display 'markup' 'strip'⎕XML eg
```



```
      ]display 'markup' 'preserve' ⎕XML eg
```



Unknown-entity

When converting from XML, this option determines what happens when an unknown entity reference, or a character reference for a Unicode character which cannot be represented as an APL character, is encountered. When converting to XML, this option determines what happens to Esc characters (`⎕AV[8+⎕IO]`) in data.

Converting from XML	
`replace`	The reference is replaced a single '?' character.
`preserve`	The reference is included in the output data as given, but with the leading '&' replaced by `⎕AV[8+⎕IO]`.
Converting to XML	
`replace`	`⎕AV[8+⎕IO]` is preserved
`preserve`	`⎕AV[8+⎕IO]` is replaced by '&'

Outer Product: {R}←X∘.gY

g may be any dyadic function. The left operand of the operator is the symbol ∘. X and Y may be any arrays whose elements are appropriate to the function g.

Function g is applied to all combinations of the elements of X and Y. If function g returns a result, the shape of R is (⍴X),⍴Y. Each element of R is the item returned by function g when applied to the particular combination of elements of X and Y.

Examples

```
      1 2 3∘.×10 20 30 40
10 20 30  40
20 40 60  80
30 60 90 120

      1 2 3∘.⍴'AB'
 A     B
 AA    BB
 AAA   BBB

      1 2∘.,1 2 3
 1 1   1 2   1 3
 2 1   2 2   2 3

      (⍳3)∘.=⍳3
1 0 0
0 1 0
0 0 1
```

If X or Y is empty, the result R is a conformable empty array, and the operand function is applied *once* between the first items of X and Y to determine the prototype of R.

Index

A

add text as field 28
add text as method 28
add text as property 28
AltGr
  keyboard 15, 16
APLFormatBias parameter 31
APLVersion property 13

C

checksum 54, 56
ClassName property 10
collapsing outlines 21, 23, 26
Compatibility 34
component files
  checksum 54, 56
  compatibility 34
  file properties 54
  journaling 4, 56
  unicode 54, 57
compress 2
create bound file 6
Ctrl
  keyboard 15, 16

D

dyadic primitive operators
  outer product 67

E

editing classes
  classes 22
editor enhancements 20
  class treeview 24, 25, 26
  classes 22
  collapsing outlines 21, 23, 26
  expanding outlines 21, 23, 26
  function line numbers 26, 27
  line numbers 27, 29
  outlining 20, 26
  refactoring 28
  speed 29
  syntax colouring 29
  view menu 26
  workspace explorer 25
erasing objects from workspaces 51
expanding outlines 21, 23, 26
export to memory 7, 50
export workspace 6
expunge objects 51

F

fchk system function 4
file properties 54
function line numbers 26, 27
Function Tracer option 30

G

grade down 2
grade up 2

I

i-beam 47
  export to memory 7, 50
  memory manager statistics 50
  number of threads 49
  parallel execution threshold 4, 49
  syntax colouring 48
  thread synchrnisation mechanism 49
IME
  keyboard 15, 16, 18
index 3
Interoperability 34

J

journaling 4, 54, 56

K

keyboard layouts
  AltGr 15, 16
  Ctrl 15, 16
  IME 15, 16, 18

L

line numbers......................................26, 27, 29
lines_on_functions parameter.........................27

M

markup 63
matrix multiplication 2
memory manager statistics 50
monadic primitive functions
    table...46
multi-core 4
multiple cpus 4
multi-threading 4

N

netcontrol object.......................................8, 10
number of threads..49

O

outer-product operator.............................32, 67
outlining 20, 26

P

parallel execution 4
    number of threads.....................................49
    parallel execution threshold.......................49
    thread synchronisation mechanism............49
parallel execution threshold.......................4, 49
primitive operators
    outer product 67
product
    outer...67
prototype 67

R

RadiusMode property....................................14
refactoring 28

S

SALT
    Load command 32
set functions... 3
size of objects 58
small-range array 2
sort 2
SPICE 5
standalone executable 6
syntax colouring 29, 30, 48

T

table function 46
thread synchronisation mechanism.................49
tracer enhancement
    syntax colouring 30
TranslateTable parameter 18
transpose 3
treeview 24, 25, 26

U

unicode 54, 57
unkown-entity..63
user commands 5

V

value tip window 29
view menu 26

W

WantsSpecialKeys parameter 18
whitespace 60

X

xml convert..59
    markup 63
    unkown-entity...63
    whitespace 60

DYALOG APL

Dyalog Ltd
South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom
www.dyalog.com